AI Coding Agents, Messaging Platform Integration, and Stateful Developer Automation: A Survey

Tripti Singh[1], Dolly Verma[2], Nidhi Sharma[3]
[1]*M. Tech Student, Department of CSE, RSR Rungta College of Engineering and Technology, C.G.*
[2,3]*Assistant Professor, Department of CSE, RSR Rungta College of Engineering and Technology, C.G.*

***Abstract*—In the past three years AI coding agents have improved a lot, moving from function-level code completion to repository-wide multi-step task execution. However, the tooling for agent access, session management, and developer feedback has not grown at the same speed. To identify the existing limitations, this survey reviews three related areas: AI coding agents and their evaluation benchmarks, agent orchestration frameworks, and messaging bot platforms. When these research areas are considered together, it becomes clear that they are still explored separately. Coding agents are powerful but terminal-bound, orchestration frameworks manage state but assume local invocation, and messaging bot frameworks reach developers on the right channels but execute no complex logic. None of the existing systems combines remote messaging access with stateful, project-aware coding agent execution and structured post-run feedback. Closing these gaps could lead to more practical and effective developer automation tools in the future.**

***Index Terms*—AI coding agents, survey, messaging platforms, session management, orchestration frameworks, developer productivity, stateful automation**

I. INTRODUCTION

AI coding agents have come a long way from providing simple code suggestions. They can now understand an entire project, edit files in different directories, and complete multi-step development tasks with only minimal guidance from developers. Tools like Claude Code [1], Cursor [2], and Codex [3] are now capable of working across entire projects and performing practical software engineering tasks. Benchmarks like SWE-bench [4] provide reproducible proof of progress.

Although AI coding agents have improved significantly, they are still accessed through a terminal or an IDE plugin. However, software teams usually communicate and manage their work through messaging platforms. Bug reports arrive in Slack. Code review notes appear on GitHub. When developers want to use an AI coding agent to handle these tasks, they usually have to switch to a terminal, run the agent manually, and then copy the results back to the messaging platform. This extra effort interrupts their workflow and makes it less convenient to use AI coding agents as part of their daily development process.

A second problem is the lack of persistent sessions. Most messaging interfaces treat every request as a new task. Each message starts fresh with no record of which project was active, what the agent changed, or how to connect a follow-up question to earlier work. For real software tasks debugging, refactoring, code review it is a serious limitation.

This survey reviews three related research areas: AI coding agents, agent orchestration frameworks, and messaging bot frameworks. It examines their progress and identifies the remaining research gaps. Section V highlights the remaining research gaps.

II. AI CODING AGENTS AND BENCHMARKS

A. Early Code Generation

Chen et al. [3] showed that large language models could generate code for function-level tasks through Codex. GitHub Copilot [5] translated this into everyday software development. Ziegler et al. [6] found that Copilot improved developer productivity on code completion tasks. AlphaCode by Li et al. [7] showed good results in competitive programming. However, these early systems focused only on code completion. They could not understand an entire codebase, plan multi-step tasks, or edit multiple files.

B. Repository-Level Agents

A newer generation of agents works at the repository level. Claude Code [1] can understand a codebase, plan tasks, and edit multiple files. Cursor [2] provides similar capabilities within an IDE. SWE-agent [8] introduced an agent-computer interface that helps language models access files and edit code. Devin [9] showed that a dedicated coding agent could resolve real GitHub issues. SWE-bench [4] provides real GitHub issues to test whether agents can reproduce the correct fixes.

C. What These Systems Share

These systems are designed for developers working on a local machine. The developer must have the repository on their machine, the agent CLI installed, and a terminal open. None allow developers to run the agent from a remote messaging platform or send results back to messaging thread. This limitation is not addressed in the surveyed studies.

III. AGENT ORCHESTRATION FRAMEWORKS

A. Reasoning and Tool Use

ReAct [10] showed that agents perform better on complex tasks when they alternate between reasoning and taking actions. Instead of completing a task in one step, the agent follows a cycle of thinking, acting, observing the result, and then deciding what to do next. This approach became a common design pattern in many agent frameworks because it helps agents make better decisions and makes their behavior easier to understand and improve.

B. Multi-Agent Systems

AutoGen [11] expanded the idea of a single AI agent by allowing multiple specialized agents to work together. These agents communicate by exchanging messages and divide complex tasks based on their different roles. Although AutoGen manages communication during a task, it does not explain how conversation history or user information is saved for future interactions.

C. Pipelines and Memory

LangChain [12] became one of the most popular frameworks for developing LLM-based applications. It provides tools for managing prompts, using external tools, storing memory, and retrieving information. LlamaIndex [13] makes it easier to organize and find data. Both leave the question of what to persist and under what key to the application developer. Both frameworks leave it up to developers to decide what information should be stored and how it should be identified. MemGPT [14] introduced a different approach by treating the LLM's context window as temporary memory and using memory management techniques to support tasks that require long-term context.

D. What These Frameworks Do Not Address

Orchestration frameworks define how an AI agent performs its tasks. However, they do not explain how the agent receives tasks from external sources such as messaging platforms or how the results are sent back to users.None of the surveyed frameworks provide built-in support for this type of communication. As a result, if a user returns later to continue a conversation, the previous context usually has to be provided again.

IV. BOT FRAMEWORKS AND MESSAGING INTEGRATION

A. Conversational Bot Platforms

Dialogflow [15] and Rasa [16] are two popular frameworks for building chatbots. Both provide features such as natural language understanding, intent recognition, and conversation management. However, they lack functionalities that support advanced agent capabilities, including performing coding operations, launching subprocesses, and detecting any file modifications after performing a particular operation.

B. Platform-Specific SDKs

The Slack Bolt SDK [17] and Microsoft Bot Framework [18] offer capabilities for developing chatbots for their individual platforms. But they are both platform-specific solutions. This means that a bot developed for Slack will not work for Microsoft Teams and vice versa. In other words, the developer will have to write individual bots for every platform.

C. Developer Tooling Bots

CI/CD bots research [19] demonstrated that developers appreciate chatbots providing fast and relevant feedback. Summaries of the modified files proved to be more useful than file code diffs and general status reports. GitHub bots research [20] also

revealed that developers would accept automated replies for typical activities. However, those bots cannot keep the history of the conversation and perform only predefined actions. No bot can execute generic AI code generation agents and retain context of the previous conversation.

D. The Missing Integration

There is no capability in the surveyed bot frameworks to write code agents. Bot frameworks are supposed to deal with messages, whereas code agents execute code. But there is no capability in any of the surveyed systems that could allow you to assign a task from a chat to a particular project and code agent and receive the results back to the chat conversation.

V. GAP ANALYSIS

Gap 1 - Remote messaging access: None of the discussed platforms enable developers to execute a coding agent right from the messaging application and get their output there in the same communication thread. The coding agent systems need terminals, and the bot systems cannot perform tasks of coding agents.

Gap 2 - Cross-session project context: None of the tested systems keeps the information about the project that has been connected to the dialogue. Therefore, users will need to reconnect their project every time. Extension of the memory in the current session is possible using MemGPT, however, the conversation cannot be connected to the project folder.

Gap 3 - Structured post-run change feedback: None of the surveyed systems provide a clear summary of the files changed after a coding agent finishes a task. This makes it harder for users to review the results through a messaging platform.

VI. DISCUSSION

There are three gaps that are related. If developers cannot use a coding agent from a messaging platform (Gap 1), then they will not have problems with the preservation of the project context (Gap 2) and seeing what changes were made (Gap 3). On the other hand, when Gap 1 is solved, then developers start to experience Gaps 2 and 3 immediately.

The right way would be to consider all of these issues as a whole problem and not as separate issues. Preserving the state of the session needs just some storage of the necessary information about each chat: the name of the active project, the name of the used coding agent, and maybe the session id if the agents have session resuming capabilities. The same can be said about the gap with change reporting because it needs just comparing the files of the project before and after the task completion by the agent.

Security is another critical factor. When designing systems which enable users to launch tasks via a messaging platform, there should be security guarantees that the only users who will launch any particular task and gain access to the specified project will be the authorized ones. One way to go about this is by having authorized user IDs.

Developer context research [21] has discovered that developers need to invest time and effort into figuring out what they had done before when coming back from a break. A system that saves project context and restores it upon returning to work can alleviate this problem. Distributed teams research [22] has also revealed that messaging platforms are popular for team communication and are a good choice for receiving the results of coding agents.

VII. CONCLUSION

The survey covered AI coding agents, agent orchestration systems, and messaging bot frameworks. All three have made substantial advances in their respective fields. AI coding agents can accomplish practical software engineering problems on a large codebase. Agent orchestration systems offer functionality for reasoning, memory management, and coordination between several agents. Messaging bot frameworks enable communication through the very platforms which developers currently use.None of the above frameworks possesses all three functionalities, i.e., being able to receive the problem from the messaging platform, execute a coding agent on the right codebase while keeping the context of the current session and giving a summary of the changes introduced. Thus, these three shortcomings form an interesting field for further research.

REFERENCES

[1] Anthropic, Claude Code: Agentic Coding in the Terminal, Technical Rep., 2024.

[2] Anysphere Inc., Cursor: The AI Code Editor, Technical Rep., 2024.

[3] M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.

[4] C. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" arXiv preprint arXiv:2310.06770, 2023.

[5] GitHub Inc., GitHub Copilot: Your AI Pair Programmer, Technical Rep., 2022.

[6] A. Ziegler et al., "Productivity assessment of neural code completion," in Proc. ACM SIGPLAN Int. Workshop on Machine Learning and Programming Languages (MAPS), 2022.

[7] Y. Li et al., "Competition-level code generation with AlphaCode," Science, vol. 378, no. 6624, pp. 1092–1097, 2022.

[8] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering," arXiv preprint arXiv:2405.15232, 2024.

[9] Cognition AI, Devin: The First AI Software Engineer, Technical Rep., 2024.

[10] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. Int. Conf. Learning Representations (ICLR), 2023.

[11] Q. Wu et al., "AutoGen: Enabling next-generation LLM applications via multi-agent conversation," arXiv preprint arXiv:2308.08155, 2023.

[12] H. Chase, LangChain: Building Applications with LLMs Through Composability. GitHub Repository, 2022.

[13] J. Liu, LlamaIndex: A Data Framework for LLM Applications. GitHub Repository, 2022.

[14] C. Packer et al., "MemGPT: Towards LLMs as operating systems," arXiv preprint arXiv:2310.08560, 2023.

[15] Google LLC, Dialogflow: Build Natural and Rich Conversational Experiences. Google Cloud Documentation, 2023.

[16] A. Bocklisch, J. Faulkner, N. Pawlowski, and A. Nichol, "Rasa: Open-source language understanding and dialogue management," arXiv preprint arXiv:1712.05181, 2017.

[17] Slack Technologies, Bolt for JavaScript: A Framework for Building Slack Apps, 2023.

[18] Microsoft Corporation, Bot Framework SDK. Microsoft Documentation, 2023.

[19] D. Hilton et al., "Usage, costs, and benefits of continuous integration," in Proc. 31st IEEE/ACM Int. Conf. Automated Software Engineering (ASE), 2016.

[20] T. Wessel et al., "How to not get rejected: The impact of GitHub bots," in Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME), 2018.

[21] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in Proc. 28th Int. Conf. Software Engineering (ICSE), 2006.

[22] C. Bird et al., "Does distributed development affect software quality? An empirical case study of Windows Vista," in Proc. 31st Int. Conf. Software Engineering (ICSE), 2009.